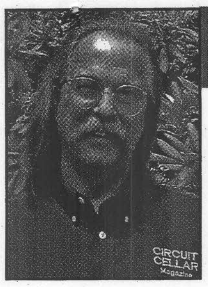


by Jeff Bachiochi

Can You Speak CAN? (Part 1)

The Newest CAN Modules

In this two-part article, Jeff introduces you to some of the newest CAN modules on the market. Get ready to begin your own CAN-based system.

"The working day shall last eight hours." What fool was responsible for that gem? Actually, it was the automotive engineer Robert Bosch (1861–1942) in a speech on work rules back in 1906. The workweek at that time consisted of six 10- to 12-h workdays. Bosch wanted to bring relief to his workers. We haven't seen much change since then. In fact, for those of us who work for ourselves, the day often lasts longer than eight hours.

Today, most of us associate the Bosch name with automotive systems. The automotive technology division of the Bosch Group has continually been at the forefront of system improvements. It specializes in fuel injection and management, chassis systems (e.g., antilock brakes and electronic stability), and energy and auxiliary systems (e.g., starters and alternators). And let's not forget Blaupunkt audio gear and tons of semiconductors and control units.

The result of the Bosch dynasty has been the proliferation of electronics in automotive design. The growth in copper wiring harnesses has exploded like subway lines extending deep into growing suburban communities. To reduce copper usage and the added weight to a vehicle, the Bosch team pared down the automobile's unwieldy umbilical cord by replacing it with an interconnecting bus to time-share control and status information. This was a huge change in the way engineers thought about system design. Today, many of the new vehicles on the market have multiple buses.

CONTROLLER AREA NETWORK

It's common to reference the seven-layer open system standard (OSI)—developed by the International Organization for Standardization (ISO) in 1984—when discussing a network (see Figure 1). Each layer defines a stage that data passes through when traveling from one device to another over a network.

The seven layers are separated into two sets: application and transport. Specific protocols may combine multiple layers as in the CAN protocol. The CAN specification (ISO11898) combines the data and physical layers of the OSI. The data link sublayer has two responsibilities: logical link control (LLC) and media access control (MAC). Together they construct the message format. The physical sublayer specifies the physical and electrical characteristics of the bus used to transmit message characters between nodes.

The hardware implementing the CAN bus may be electrical (single wire, twisted pair, or even wireless) or optical (fiber), and isn't defined by the specification. However, the physical signal encoding (non-return to zero, or NRZ), bit timing, and synchronization is totally defined. NRZ encoding defines a 0 bit as a logic 0 level and a 1 bit as a logic 1 level (see Figure 2). This differs from Manchester encoding, where a 0 bit is a logic 0-to-1 change, and a 1 bit is a 1-to-0 change.

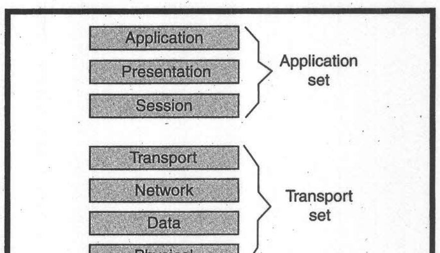


Figure 1—*These seven layers represent a conceptual framework for implementing communications across a network. Many protocols will cover multiple layers. The CAN protocol covers both the data link and the physical layers as defined by ISO11898.*

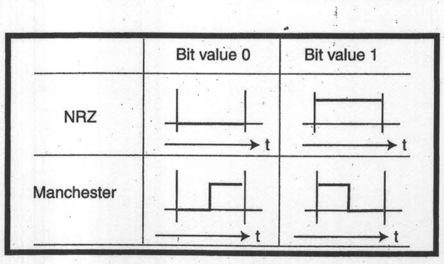


Figure 2—*The two most widely recognized ways a bit state can be defined is by a constant logic level at some sample time (e.g., a non-return to zero (NRZ)) or a change of state (rising or falling edge) at some sample time (e.g., Manchester coding). Although Manchester offers convenient edges to synchronize on, it can require twice the bandwidth of NRZ.*

Manchester encoding has the advantage of providing at least one edge/bit from which periodic clock synchronization can be extracted. NRZ encoding can pass more data per transition (which usually limits bandwidth) so it can be more efficient. As you can imagine, NRZ data consisting of numerous 0 or 1 bits would fail to change states. All asynchronous communicating devices must be able to synchronize their receiver clocks to the data being transmitted. Without any edges (changes in data state), this can't be accomplished.

Bit stuffing is implemented to guarantee an edge every 5 bits. The bit-stuffing rule simply defines that if any transmitted data remains constant for five bit times, an extra bit of the opposite polarity is automatically added to (stuffed into) the transmitted data. At the other end, if received data remains constant for five bit times, the next bit is tossed out.

Asynchronous serial data is normally unidirectional in nature. A single source provides data. The CAN protocol uses carrier sense multiple access with collision detection (CSMA/CD). Every CAN node must monitor the bus for no activity before transmitting. This allows any node to act as a producer of data, while other nodes are the consumers of data.

Collision detection allows nodes to perform nondestructive bitwise arbitration. This arbitration requires all of the transmitters to listen to the bus while they're transmitting. Think of logic 1 as recessive and logic 0 as dominant. If a node outputs a 0 (dominant), the bus will reflect this fact (read as a 0). If a node outputs a 1 (recessive), the bus will reflect this fact (read as a 1) unless some other node is dominating by outputting a 0. The CAN protocol requires a recessive node to back off and give way to the dominant node.

For this to work properly, the bit's sample point must be positioned correctly within the bit. In normal serial communication, a bit is sampled somewhere around half of the bit time. A propagation delay time is added to the CAN's sample point calculation (see Figure 3). The propagation delay time allows for a dominant state from a distant node to reach a recessive node. So, basically, a sort of hierarchy of importance is established on the bus by assigning lower dominant values to those nodes with higher importance.

The frame sends data. The CAN protocol defines four frame types: data (Here is my data), remote (Please send this data), error (I detected this error), and overload (Hold on, I'm not ready). Figure 4 shows the arrangement of data within each of the four frame types.

The data frame has the remote transmit request (RTR) bit cleared, and the remote frame has the RTR bit set. Otherwise, the format of each is identical. The areas of concern to the user within these frames are the arbitration field, the control field, and the data field. While the 11-bit identifier has an inherent bus priority associated with it, it is also used by a receiving node's message filtering to determine if the message needs to be processed or neglected. Note the identifier extension (IDE) bit located right after the RTR bit. If this bit is set, the 18-bit extended identifier is expected prior to the control field.

In systems where the 11-bit identifier isn't sufficient, the arbitration field can be expanded to 32 bits if desired. This would be true for both the data and remote frames. Note that arbitration between a node using the standard data frame and another node using an extended data frame with the same 11-bit identifier is won by the standard data frame (IDE=0 is dominant).

The control frame's data length code bits indicate the number of data bytes

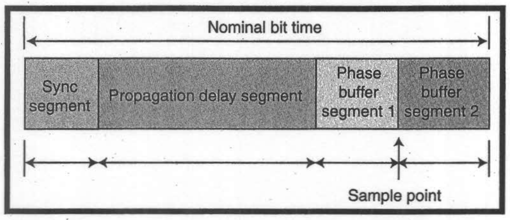


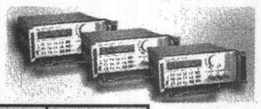
Figure 3—*In typical asynchronous communication, a bit's logic level is sampled somewhere around half (in the center of) the bit time. The nominal bit time of an asynchronous CAN communication uses a slightly different calculation for determining the sample point. There is a propagation delay associated with the sample point such that a node at one end of the bus must be able to respond to a node at the opposite end of the bus before the sample time.*

The New Rabbit
Designed For The Most Demanding Embedded Control Applications
• Speeds up to 58.98 MHz
• Ethernet with royalty-free TCP/IP
• Supports 16-bit memory devices
• 8 Channel 12 bit A/D
• Up to 6 serial ports
• Large NAND Flash
• 512K Flash
• 40 GPIO
New! Next Generation Rabbit 4000
RABBIT® 4000 AT58206-UL1T DOE4946AA 0533
RCM4000
RCM4100
Kits From $259
RABBIT Semiconductor
Order Online At rabbit4000.com
06160

Circuit Specialists, Inc.

Programmable DC Power Supplies

•Up to 10 settings stored in memory
•Optional RS-232 interface
•Low output ripple & noise
•High resolution at 1mV



Model	CSI3644A	CSI3645A	CSI3646A
DC Voltage	0-18V	0-36V	0-72V
DC Current	5A	3A	1.5A
Power (max)	90W	108W	108W
Price	**$199.00**	**$199.00**	**$199.00**

Full Details on Web Site!

34 Channel Logic Analyzer

500MHz Timing Mode Sample Rate
$389.00
***Full Details on Web Site* Item# LOGICPORT**

A Sophisticated Scope Adaptor
200MHz USB DSO
$749.00
Full Details on web
Item#: 200DSO

A Price Breakthrough!

www.CircuitSpecialists.com
800-528-1417 / Fax:480-464-5824

(up to eight) that follow in the data field. A 15-bit CRC field follows the data field to aid in determining transmission errors. After the CRC field, there's an ACK bit. Receiving nodes confirm correct reception by applying a dominant 0 in this bit. The protocol handles all of the packet building and checking. You supply information for the identifier, control, and data fields, and the CAN protocol does the rest.

Between each data and remote frame is a mandatory inter-frame space, which is essentially bus idle time. However, it also allows the two other frames to be used. They're pretty much the same in format, as you can see in Figure 4. Any node that detects an error in reception can indicate such by waving dominant bits onto the inter-frame space. This in itself is an error in the protocol, and all nodes will respond with error frames. The sender can take the appropriate action (retransmit). The overload frame will do much the same thing. Although its purpose might be to delay frames, this is essentially a violation of the bit-stuffing rule.

CAN TRANSCEIVERS

The CAN specification describes the recessive/dominant communications between nodes. It doesn't specify the particular parts used to accomplish this. This circuitry can vary depending on the transmission medium (electrons versus photons). You may be familiar with RS-485's use of a differential twisted pair bus. These drivers can handle long cable runs, and they offer immunity to noisy environments because of the differential twisted pair. The problem with this hardware is that it doesn't support multiple nodes outputting opposing states on the bus. So, RS-485 drivers can't be used in a CAN environment.

Figure 5 (p. 76) shows how a typical CAN transceiver differs from the RS-485 drivers. A logic 1 and 0 input to a RS-485 device drives output levels hard in opposite directions to a 5-V differential voltage. In a CAN device, a logic 1 input doesn't drive the output, but it allows both outputs to float to approximately 2.5 V, the bus's nominal bias level and recessive state. A logic 0 on the input drives the CANH output high and the CANL output low (the dominant state).

The slew speed of the outputs can be controlled via the RS pin. You may want to adjust this slope control based on line length and speed to optimize noise rejection.

Because safety is a concern on a bus where any transceiver could cause a total failure of communications, the CAN transceiver has built-in dominant failure detection. Let's say a node failure causes the TXD to get hung low, leading to an extended dominant state. After approximately 1.25 ms, the transceiver will disable itself as long as the TXD remains low. Power-out and brown-out protection disables the transceiver as well. When using this twisted

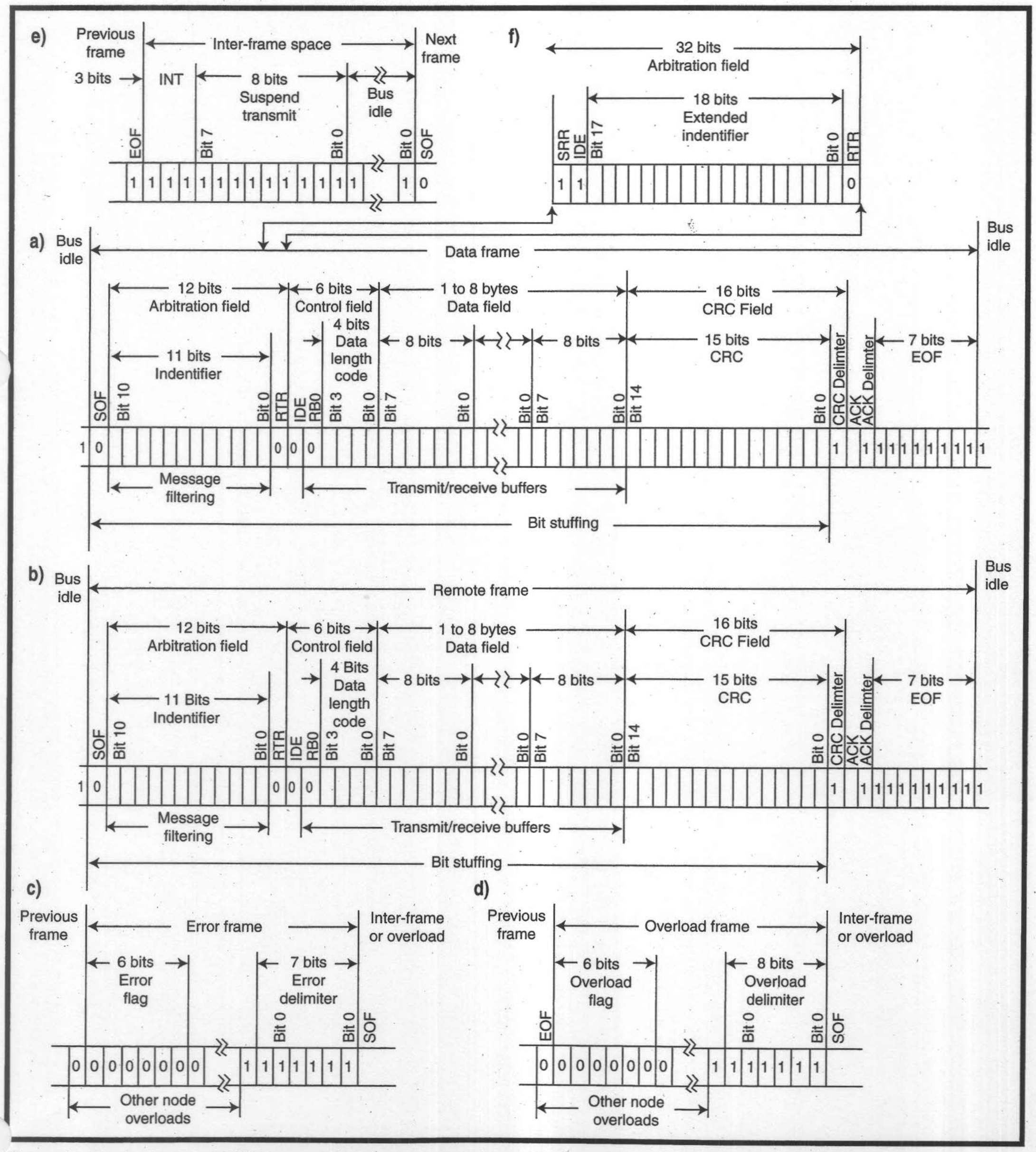


Figure 4—*The four types of CAN frames defined in this protocol are: data frame **(a)**, remote frame **(b)**, error frame **(c)**, and overload frame **(d)**. An inter-frame space is expected after each data and remote frame **(e)**. This provides time for error and overload frames to be recognized. The extended identifier is optionally supported by CAN 2.0B **(f)**.*

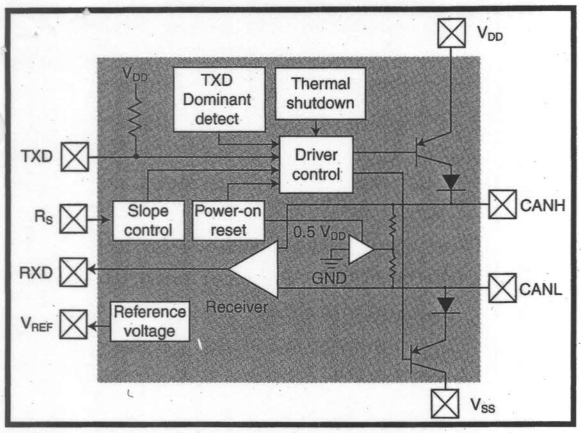


Figure 5—*While an RS-485 transceiver forces its differential outputs to logic states under both data conditions, the CAN transceiver can only pull its outputs in one direction. A logic high input disables any drive, allowing the outputs to float to half of V_{CC} (recessive state). A logic low input drives the outputs to their high and low output levels (dominant state). This enables any node to dominate (win bus arbitration) another node that's outputting a recessive state.*

pair interface, the bus should be terminated to 120 Ω at each end.

Although no standard CAN connectors are specified, labeling and pin assignments are suggested for various connector styles. Several connector styles are recommended: nine- and 15-pin D-sub, RJ-10 and RJ-45, five-pin SIP, five- to 12-pin round connectors, and the IEEE 1394 (FireWire) connector. The style you choose may depend on the distance between nodes, environmental concerns, or simply what's necessary to mate to an existing bus.

Assuming all this bottom-layer stuff has been implemented, what is actually necessary to communicate using this lower layer hardware?

TRANSMISSION & RECEPTION

Multiple manufacturers support CAN communications, even in low-pin-count microcontrollers like Atmel's AT89C51 and Freescale's MC68HC08. Microchip took an interesting tack in implementing a CAN controller. It developed a CAN peripheral off microcontroller that uses a SPI to interface to any microcontroller. Not only is this a less expensive development project, but it also has the benefit of allowing any processor to take on the CAN bus. The Microchip MCP2515 consists of a total CAN engine including two receive buffers and three transmit buffers complete with filters and masks (see Figure 6).

What are masks and filters, and how are they used? Earlier I explained that you have access to CAN data consisting of control stuff (an 11-bit identifier and 18-bit extended identifier, if used) and up to 8 data bytes (14-byte buffer). When a CAN message has been received and CRC verified, it leaves the protocol engine and is transferred to the message assembly buffer (MAB). At this point, it waits to be transferred to one of the two receive buffers.

Not every message received by a node is necessarily of interest to that node. A mechanism to reject unnecessary messages keeps the workload down to a reasonable level. This mechanism consists of a set of mask and filter registers for each receive buffer. Each mask consists of four registers. Each filter consists of four registers. These registers are associated with the 11-bit identifier field and either the 18-bit extended identifier field (IDE bit=1, if used) or the first 2 data bytes.

The mask registers identify bits of importance. Let's say that a system is made up of numerous nodes, each with two digital input bits (an On button and an Off button). One node has a digital output bit that controls a light. Every node with push buttons could send a change-of-state flag via the 11-bit identifier's LSB and data about this change. The node that controls the light's state need only pay attention to those messages that have commands from nodes with push button status information. By placing a 1 in the LSB of the identifier's position in the mask register (with all other bits as 0), you've indicated that only this bit is of any significance.

The filter registers determine the logical state of interest for any significant bits. In this case, only the 11-bit identifier's LSB in the MAB is compared to its corresponding bit in the filter register. A match of these enables the MAB's contents to be moved into the associated receive buffer. Otherwise, the message is discarded. With these simple tools, this node will receive only the messages in this buffer that may affect the state of its output. A totally different mask and filter can be used to restrict messages for the second receive buffer. It's up to you to determine from the received data if action is necessary. This data may contain light

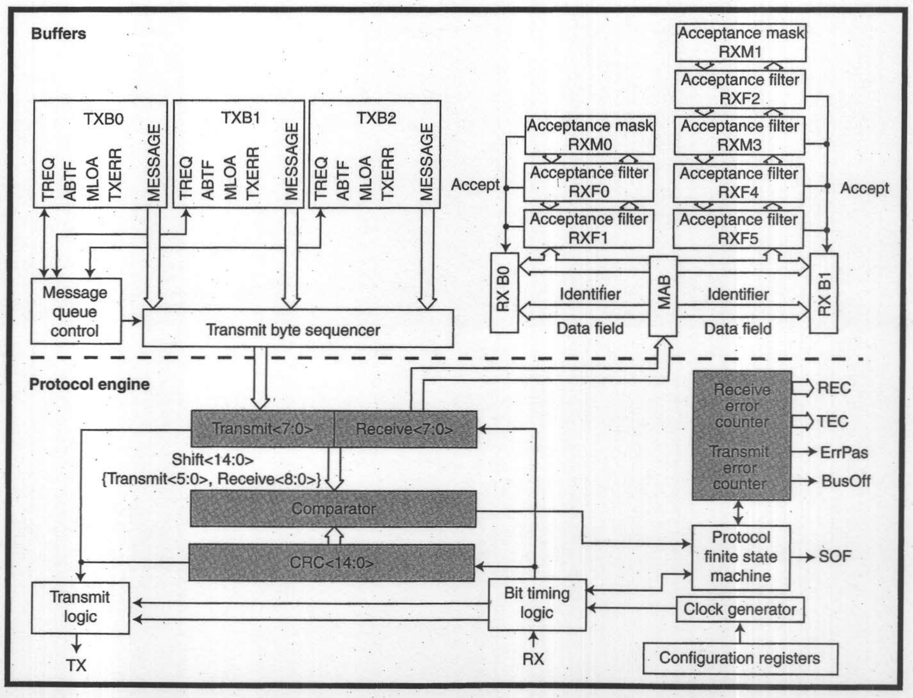


Figure 6—*The CAN controller contains two distinct areas, the data buffers and the protocol engine. Separate buffers are provided for data coming and going. Mask and filter registers help determine if a received message needs any processing.*

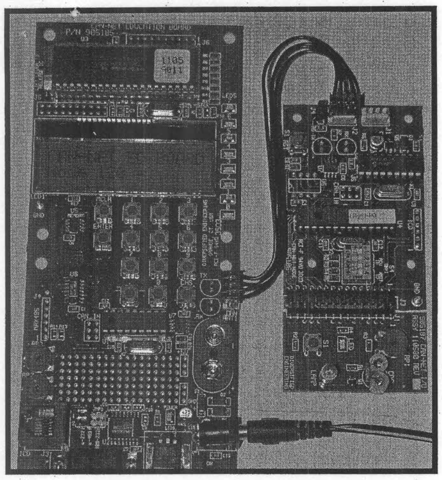


Photo 1—*CAN nodes available from Diversified Engineering provide a good platform for experimenting with or developing your own CAN-based applications. A preprogrammed CAN application (with source code) lets you investigate node communications immediately.*

level information or an instruction for a timer to keep the light on for 1 min.

Sending a CAN message doesn't involve any masks or filters. You're responsible for setting up the 14-byte buffer registers properly. This includes the identifiers, control, and data bytes. Multiple transmit buffers (three) allow messages to be cued up for transmission or each transmit buffer set to send some standard response or request. The priority of these buffers can be configured to force the order in which multiple enabled messages get sent. Even though the CAN protocol will handle errors and automatically resend messages, individual transmit buffer errors can be monitored.

CAN EDUCATION

There is nothing like playing with real hardware to shed light on those hazy areas of operation. Diversified Engineering has been supporting Microchip's PIC micros and the CAN interface for years now. It offers a number of CAN modules that you can use to develop your own CAN system and some courseware to get you started (see Photo 1). I used an oscilloscope on Diversified Engineering's Qik Start CAN-NET Education Board to look at some real CAN messages. It came preprogrammed with a simple CAN-NET application.

Photo 2 shows bit stuffing used by the CAN protocol engine to ensure that some logic level transitions take place after sending five sequential data bits of the same polarity (an 11-bit identifier of 0x000 is being sent). Although the actual length (total message bits) of a CAN transmission may vary as a result of bit stuffing, a maximum data length of 8 bytes ensures that CAN messages are short. This particular request message requires less than 500 µs of bus time.

In Photo 1, the larger node incorporates a bunch of I/O (a keypad, LCD, and a serial port) in its CAN implementation. In Stand-Alone mode, the LCD can display the 2-byte data associated with any of the identifiers selected via the keypad. The serial port outputs all CAN bus messages to a PC running Diversified Engineering's free CANMan bus monitor program. This program allows you to set mask and filter values to limit which messages the monitor receives. Also, you can send a CAN message directly from the monitor. This is great for when you're developing CAN modules.

Diversified Engineering's CAN modules make use of Microchip's stand-alone MCP251*x* CAN controller. The success of this viable CAN peripheral naturally led to the inclusion of this controller as an internal peripheral to some devices in the PIC18F series of microcontrollers.

CAN REVISIONS

Like any good specification, the CAN protocol continues to evolve. Version 2.0 specifications are divided into parts A and B. There is one basic difference between these two parts. Part A is the original protocol that uses the standard 11-bit identifier followed by a control field beginning with two reserved bits. Part B redefines the first reserve bit as an IDE indicator and goes on to define the extended identifier field. Originally a reserved bit, early CAN engines should accept a frame finding this reserved bit recessive. However, the extended identifier that follows won't fit the expected format. Therefore, CAN devices may be rated according to their adherence to the present specifications.

The minimum acceptable rating would be CAN 2.0A (i.e., conforms to the protocol's original 11-bit identifier). A CAN 2.0B rating may be active or passive.

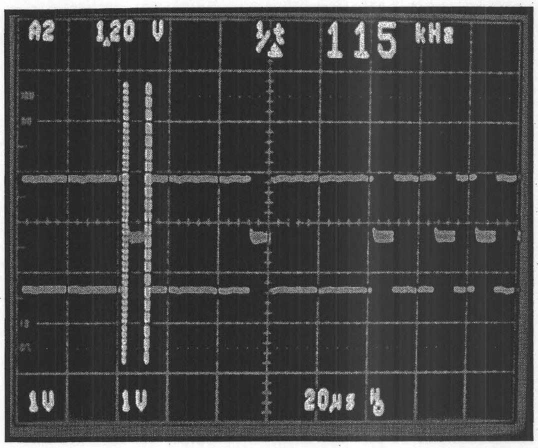


Photo 2—*The sequence seen here is the beginning of a CAN message with an identifier of 0x000. Bit stuffing is used to ensure an edge at least once every five bit times. In this oscilloscope shot, you can see that beginning with the SOF. A 0x000 identifier has a bit of (in this case) a recessive state stuffed into the data transmission after each five unchanging (in this case dominant) bits. The CAN protocol engine automatically removes these at the opposite end. It shows a recessive bit stuffed after the five dominant bits (SOF and ID bits 10 through 7). Another recessive bit is stuffed after the five dominant bits (ID bits 6 through 2). One more recessive bit is stuffed after the five dominant bits (ID bits 1 through 0, RTR, IDE, and RB0).*

CAN 2.0B passive devices wouldn't choke on extended identifier messages (cause message errors); however, they wouldn't process them. A CAN 2.0 active device fully supports messages using standard and extended identifiers. It's recommended that all new devices conform to CAN 2.0B passive as a minimum. ■

Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for Circuit Cellar *since 1988. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circuitcellar.com.*

RESOURCES

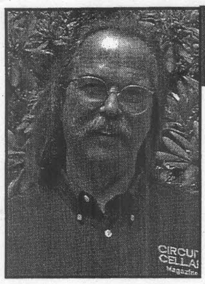
Bosch Group, CAN Specifications, www.semiconductors.bosch.de/en/20/can/index.asp.

CAN in Automation (CiA), www.can-cia.org.

SOURCES

Qik Start CAN-NET Education Kit
Diversified Engineering & Manufacturing, Inc.
www.diveng.net

MCP2515/2551 CAN devices
Microchip Technology, Inc.
www.microchip.com



FROM THE BENCH

by Jeff Bachiochi

Can You Speak CAN? (Part 2)

A Look at CANopen

Last month, Jeff introduced some of the new CAN modules on the market. This month, he describes the advantages of using the higher-level CANopen protocol.

Whether you've chosen to use the CAN communication interface for its multimaster capability, inherent noise immunity, transparent transmission handling, priority messaging, or any combination of these and other advantages, you end up with a serious question left unanswered. While the CAN interface defines the bottom tiers of the seven-layer OSI standard for networks, this hardware will most likely need some embedded code to define the upper tiers of the standard, no matter how simple it is. Even in the simplest of systems, you'll need to define the data's form when it is transmitted.

One of the advantages of the CAN bus is the ability of a node to simply pass its data to the bus. Because all the nodes can monitor this information, the data does not have to be passed to every node that might use it. In fact, the transmitting node doesn't have to know who else is out there. Eliminating node-to-node traffic reduces bandwidth requirements.

If your nodes will be used with other CAN nodes, all bets are off. You'll need to adhere to the rules already implemented by the nodes on the bus. The bottom line is that if you have total control, go ahead and define your own upper-level protocol. However, you might want to consider using one of the already established CAN protocols.

HIGHER-LEVEL PROTOCOLS

CANKingdom was developed specifically for machine control, while other protocols are more focused on factory automation. Among other things, CANKingdom supports hot swaps in a safe manner. It also supports dynamic change of identifiers, a global clock, module identification by EAN/UPC code, Std and Ext identifiers, and a hardware-only limited number of modules in a system. CANKingdom allows other protocols to be integrated into a CANKingdom system, usually with a loss of system performance.

In the early 1990s, the Society of Automotive Engineers (SAE) Truck and Bus Control and Communications Subcommittee started to develop a CAN-based application profile for in-vehicle communication in trucks. Today, the SAE's J1939 standards family is the preferred HLP for equipment used in numerous industries (e.g., marine and construction).

DeviceNet is used mainly in industrial applications (factory automation in particular). It's a low-cost communications link used to connect industrial devices (e.g., limit switches, photoelectric sensors, and operator interfaces) to a network and to eliminate expensive hard wiring. The Open DeviceNet Vendor Association (ODVA) developed the internationally standardized DeviceNet specifications. When you buy the DeviceNet specification, you receive an unlimited royalty-free license to develop DeviceNet products.

CANopen was developed as a standardized embedded network with highly flexible configuration capabilities. It was originally designed for motion-oriented machine control networks such as material-handling systems. Now it's used in medical equipment, off-road vehicles, maritime electronics, public transportation, and building automation.

CANopen provides standardized communication objects for real-time data (process data objects, or PDO), configuration data (service data objects, or SDO), special functions (timestamp, sync message, and emergency message), and network management data (boot-up message, NMT message, and error control). Some CANopen specifications (draft standard) are free. CAN in Automation (CiA) membership is required to download work drafts.

It is reassuring to know that the hardware you may have designed can be used with any protocol, provided the embedded processor you've chosen has the application space to support the protocol of choice. Using a small microcontroller

COB ID	Communication objects	Comment
0x000	NMT Services	From NMT master
0x080	Sync message	From SYNC producer
0x081–0x0FF	Emergency message	From nodes 1–127
0x100	Timestamp message	From timestamp producer
0x181–0x1FF	First transmit PDO	From nodes 1–127
0x201–0x27F	First receive PDO	For nodes 1–127
0x281–0x2FF	Second transmit PDO	From nodes 1–127
0x301–0x37F	Second receive PDO	For nodes 1–127
0x381–0x3FF	Third transmit PDO	From nodes 1–127
0x401–0x47F	Third receive PDO	For nodes 1–127
0x481–0x4FF	Fourth transmit PDO	From nodes 1–127
0x501–0x57F	Fourth receive PDO	For nodes 1–127
0x581–0x5FF	Transmit SDO	From nodes 1–127
0x601–0x67F	Receive SDO	For nodes 1–127
0x701–0x77F	NMT error control	From nodes 1–127

Table 1—*This is the predefined list of CANopen COB IDs. Functions that include a node (1–127) have individual COB IDs for the node. For instance, an emergency message from node 5 has the COB ID 0x085 (0x080+5).*

may eliminate the possibility of implementing certain protocols. If a protocol's minimum requirements can't be supported based on the available memory (both code and data space), you're out of luck.

DESIGNING FOR CANopen

Last month, you learned that the CAN message data frame was basically made up of an 11-bit identifier field, a control field (which among other functions could extend the identifier to 29 bits), a data field (containing up to 8 data bytes), a 15-bit CRC checksum, and a message acknowledge. An interframe space provides time for overload and message frames to interact with data frames. The CAN protocol engine handles this in a way transparent to the user; however, it's handy to know how this works because the frame fields are part of the software interface the higher level protocol must deal with to be able to communicate through the CAN bus.

How a higher-level protocol makes use of the CAN message's identifier field is really the basis of the protocol. Remember that any node may produce a CAN message frame at any time. Frames that occur simultaneously must win the bus through arbitration. The message sent with the lowest (value) identifier has the highest priority and gets to finish its message frame. The loser must wait and try again. So, the value of the identifier indicates its importance regarding message priority.

The CANopen standard uses a connection object (COB) ID to define message functions. Table 1 shows the predefined set of COB IDs. The CANopen standard has room for 127 nodes. Although data traffic doesn't need to know which nodes are on the bus because they generally don't use node-to-node communications, each node does have its own COB ID that can be used to identify it. Let's take a look at each of the COBs to see how they're used within the CANopen protocol.

NMT SERVICES

CANopen nodes connected to a CAN bus are entities on that CAN network. Their node numbers can individually identify them. However, when necessary, all nodes can receive a network message from a network master or maintenance program used to configure nodes on the network.

As you can see in Table 1, an NMT message has a COB ID equal to 0x000 (11-bit identifier). This special message contains a data field using only 2 bytes. The first byte contains command information; the second byte identifies whom it's for. If the second byte is 0x00, it's for everyone; otherwise, the byte holds the intended recipient (nodes 1–127).

There are five available NMT commands: 0x01 (switch to Operational mode), 0x02 (switch to Stopped mode), 0x80 (switch to Preoperational mode), 0x81 (reset), and 0x82 (reset communications). You can see how these commands might be used to put a node (or the entire network) in a particular state. A CANopen node must be capable of recognizing all NMT commands and following a general state model (see Figure 1). Because the NMT COB ID has the lowest possible value, its message has the highest possible priority.

NMT ERROR CONTROL

At the opposite end of the priority level is the NMT error control object, which has two uses. It's used as a boot message when a node transitions from initialization to preoperational modes, and it's used as the heartbeat message. In both cases, the message frame includes a single data byte containing the state (mode) of the node. Heartbeat messages can be either requested by a master node (node guarding) or, more preferably, configured to periodically broadcast its presence.

Note that the value of the state used in the message differs from the value sent by an NMT service command. It reflects the present state and not a requested one.

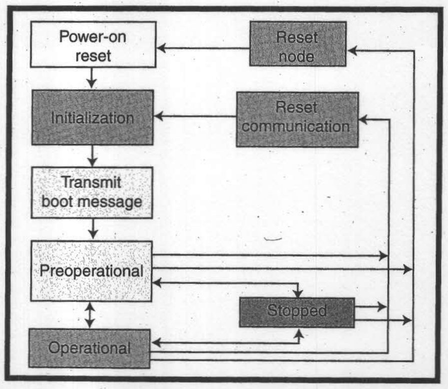


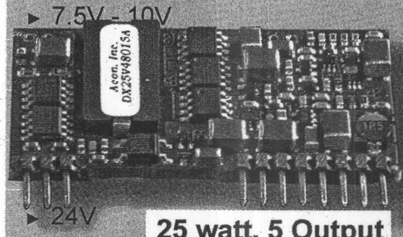
Figure 1—*A CANopen node should follow this general state diagram so that it will transition properly from an NMT service request. This provides node control via the network for configuration purposes.*

ACON™
DC-DC CONVERTERS

DC-DC CONVERTER
▾ Single Output
▸ 1.2V
▸ 1.5V
▸ 5.0V
▸ 7.5V - 10V
▸ 24V
▸ 28V
▸ 48V
▸ 60V - 80V
▸ Dual Output
▸ Triple Output
▸ Quad Output
▸ Five Output
▸ 300 Vdc Input
▸ PoE (Power over Ethernet)
▸ Chassis Mountable
▸ ACON Converter Series
▸ Request for Custom Design
▸ Technical Documentation

© ACON, INC., 2005

See 2,000 detailed model specifications www.ACON inc.com



25 watt, 5 Output

Over 2,000 Models
- Open Frame & Encapsulated
- Input Range : 9 to 420 Vdc
- Output Voltage : 1.2 to 80 Vdc
- Number of Output : 1 to 5
- Output Power : 2 to 300 Watts

ACON inc
South Easton, MA 02375
Toll Free: 1-800.336.8022
Web: www.aconinc.com
E-Mail: Sales@aconinc.com

SYNC MESSAGE

A SYNC message has no data bytes associated with it. The message, which is sent by one node, indicates a point in time when all nodes (watching for it) should simultaneously sample and respond with their data. The SYNC producer knows that this data defines the data at an instant in time no matter when the data is actually received. This method of gathering data is preferable to polling nodes individually.

EMERGENCY & TIMESTAMP

You might have noticed that the emergency message uses a similar COB ID range. Although the SYNC message has no node ID associated with it, the emergency message references the node reporting the emergency. The message has at least 3 data bytes. The first 2 bytes denote a predefined error code. The third byte is a copy of the error register (located at index 0x1001, subindex 0x00). The remaining five optional bytes are manufacturer-specific. Note that an error is reported just once, unless the report is a reset of the error condition (first data byte=0x00).

A timestamp message is similar to a SYNC message. It's simply sent to all nodes indicating the time of day. A 6-byte data message consists of 3 bytes in milliseconds of the time since midnight and 3 bytes in days since January 1, 1984.

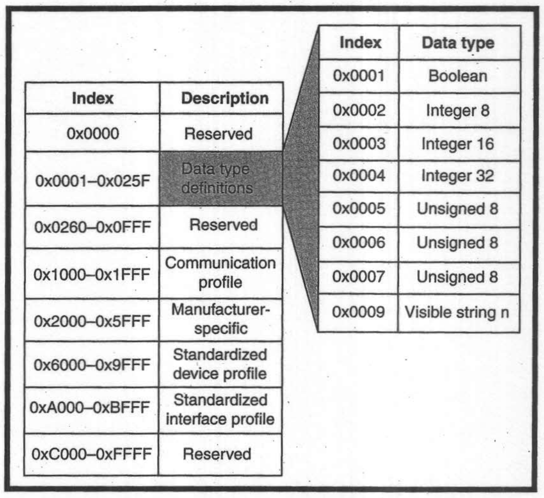
CANopen SPECIFICS

Let's get to the nitty-gritty of the CANopen protocol, which includes a method of accessing all of a node's information via the service data object (SDO) communication object. In addition to this method, it also includes a shortcut method for accessing multiple data elements in a single process data object (PDO) message. To understand these, let's back up a bit and look at the concepts involved. Although you'd begin a discussion of most higher-level protocols with this, I believe that looking at the messaging format (discussed above) will help to put the following in the proper context.

As a design engineer, it is imperative to have access to a datasheet for each part you are attempting to use. The datasheet not only describes the physical dimensions and connections, but also how it behaves electrically (or mechanically). Suppose that the temperature sensor you're using has an output of 10 mV/°C. If that part were to become scarce, you could make use of another part even if it isn't identical. For instance, a sensor with a 10 mV/°F could be used by just changing the calculations used along with the data.

The CANopen protocol uses an object dictionary in each node to describe all of the processes and communications possible with it. This consists of a table of 65,536 possible entries, each with an 8-bit subentry. This will most likely raise an eyebrow in a discussion of embedded controllers. It's important to note that every table entry is not required, and unused entries are not implemented. This significantly reduces the table size.

The object dictionary is organized in sections (see Figure 2). All dictionary entries are used to locate data associated with the entry and indicate how that data is stored (by data type). This gives each node a way of presenting information about itself via exploration. A typical dictionary might be a look-up table of 6-byte entries. Each entry would begin with a 2-byte index (dictionary object) followed



Index	Description
0x0000	Reserved
0x0001–0x025F	Data type definitions
0x0260–0x0FFF	Reserved
0x1000–0x1FFF	Communication profile
0x2000–0x5FFF	Manufacturer-specific
0x6000–0x9FFF	Standardized device profile
0xA000–0xBFFF	Standardized interface profile
0xC000–0xFFFF	Reserved

Index	Data type
0x0001	Boolean
0x0002	Integer 8
0x0003	Integer 16
0x0004	Integer 32
0x0005	Unsigned 8
0x0006	Unsigned 8
0x0007	Unsigned 8
0x0009	Visible string n

Figure 2—*The object dictionary is divided into various sections. Many data types are fixed; however, there is a mechanism in place for describing other data types like extended and complex. When a dictionary entry refers to the data type, the value present identifies the type of data being listed. The data may be a standard data type or a manufacturer-specified data type.*

by a 1-byte subindex (subobject). These 3 bytes identify a particular dictionary object. The fourth byte, the data type, describes the format of the dictionary object's data pointed to by the last 2 bytes.

The first section in the dictionary has to do with data types and deals with identifying how data is stored. There are several predefined standard data types like Boolean and 8-bit integer. When data is referred to, it has a data type associated with it. For instance, if the data type 0x02 is used, its information will be in an 8-bit integer data format. Because this first section is predefined, it isn't mandatory. Although the protocol predefines many data types, special data formats can be defined in this section. A table entry here would associate a data type value with a format description of data that is stored elsewhere.

In the second section, the communication profile defines all of the special qualities of this node (see Table 2). The first entry, the device type, references a document that defines all of the specific information necessary to use the node. The referenced document is selected from one of many predefining different device profiles, such as battery chargers, encoders, and motor drives. The error register I covered earlier is located here. This mandatory entry of data type 0x05 (1 byte) reflects the present general state of the node (0 = no errors).

The NMT entry at 0x1F80 allows a node to automatically go into Operational mode without waiting to receive an NMT command to do so. This is an important addition to the CANopen standard because many small systems may not need any network configuration.

Any node can be a producer of heartbeat messages. This is a periodic message proclaiming the node's state. Any node that keeps track of other nodes' heartbeats is a consumer. It's responsible for taking the appropriate action should a node fail to produce a heartbeat message.

Last month, I discussed the process of registering for a vendor ID. The communication profile has an entry location for this ID. It's the only mandatory entry in the identity object. The subindex 0x00 identity object entry has a data type 0x05 (1 byte) that's similar to an error register. In this case, it identifies how many subindex entries are available. It uses a data value of 1, indicating one subindex. The vendor ID uses a data type 0x07 (4 bytes). Other optional subentries available in the identity object include your product code (subindex 0x02), revision number (subindex 0x03), and serial number (subindex 0x04). Using these would include changing the value of the subindex 0x00 (number of subindexes).

Index	Subindex	Index description	Subindex name	Data type
0x1000	0x00	Device type information		0x07
0x1001	0x00	Error register		0x05
0x1008	0x00	Device name (recommended)		0x0C
0x1016	0x00	Heartbeat: Time (Only if consumer, then one entry for each producer)	Number of entries	0x05
	0xnn		Time in milliseconds for node (nn=0x01–0x7F)	0x07
0x1017	0x00	Heartbeat: Time (only if producer)	Time in milliseconds	0x07
0x1018	0x00	Identity object	Number of entries	0x05
	0x01		Vendor ID	0x07
0x1F80	0x00	NMT Startup		0x07

Table 2—*The device type (like the data type) is selected from a predefined list. These are actually documented profiles of devices that are presently going through or have previously gone through the development process as a CANopen device standard. Additional communication entries are necessary to describe the I/O for each specific device type. Only the mandatory communication entries are shown except where noted.*

Let's take a look at a generic I/O device to see which other parameters are required with this specific device type. The entry for a generic device type holds important information. As you can see in Table 3, the device type is an unsigned 32-bit entry (4-byte) entry. The two least significant bytes indicate the device profile number (in this case 0x0401, document DS401). The lower 4 bits of the next most significant byte indicate any I/O that's implemented on the device: bit 16 for any digital inputs, bit 17 for any digital outputs, bit 18 for any analog inputs, and bit 19 for any analog outputs. The most significant byte indicates special functions.

In a generic device, there is presently one special predefined option: joystick. A one indicates implementation in the respective bit position. The rest of the bits are unused bits and reserved for future use. You can determine a lot about a device by simply looking at the entry in location 0x1000 of its dictionary.

When a device supports any of the four aforementioned I/O types, additional entries are required for each I/O in the communication and in the device profile sections. The communication entries for inputs and outputs are similar and can be found at specific locations in each section. An output's definition is divided into two areas: the PDO data is packed into (beginning with the first output at location 0x1400) and how the data is mapped into (positioned within) a message (at a corresponding location of 0x1400 + 0x0200).

As in most entries, the first subindex indicates how many subentries are associated with this entry. The COD ID is at subindex 0x01. This entry also uses the most significant bit to enable/disable the uses of PDOs. (Remember that PDOs are a way for a device to pack multiple data objects into a

Index	Subindex	I/O	Index description	Subindex name	Data type
0x1400	0x00	Output	RPDO #1	Highest subindex supported	0x05
	0x01			COB ID used by PDO	0x07
	0x02			Transmission type	0x05
0x1600	0x00	Output	RPDO #1 Mapping	Number of entries	0x05
	0x01			PDO Mapping of object #1	0x07
0x1800	0x00	Input	TPDO #1	Highest subindex supported	0x05
	0x01			COB ID used by PDO	0x07
	0x02			Transmission type	0x05
	0x03			Inhibit time	
	0x04			Compatibility entry	
	0x05			Event timer	
0x1A00	0x00	Input	TPDO #1 Mapping	Number of entries	0x05
	0x01			PDO Mapping of object #1	0x07

Table 3—*A generic I/O device that implements I/O functions requires these additional communication entries. In this case, they may be digital or analog.*

single message, thereby saving bandwidth.) The last mandatory subentry indicates if the data is processed on a SYNC (which I discussed earlier) or asynchronously.

The entry at index 0x1600 holds the actual mapping of the object in a PDO. Because the longest CAN message can transfer up to 8 bytes, a PDO can handle any combination of 64 bits. The subindex entry 0x01 is 4 bytes, indicating the index and subindex of where the data comes from and how many bits are used by this entry. A second output would have its own entry at subindex 0x02.

Inputs are handled the same way as outputs beginning at the index location 0x1800. Although it's just an optional parameter and shown only in the TPDO, both inputs and outputs have the ability to periodically send out a PDO message (a broadcast of the present data in the PDO). An entry subindex of 0x05 (at 0x140x or 0x180x) has a value in milliseconds, which shows how often this should be performed.

Using the object dictionary entries, you can determine the type of device being described. You can find information about any I/O the device supports and determine how this data might be mapped into a single PDO message. Now additional information can be found in the device profile section of the object dictionary. This is the section that includes the location of the actual data (unmapped). The device profile section contains separate entries for each type of input and output.

Digital inputs require two entries. The first entry at location 0x6000 indicates how many 8-bit subentries there are for this entry. The subentries hold the actual data in 8-bit format. The CANopen protocol's flexibility is demonstrated in this section. Although it isn't shown in Table 4, the data can be accessed in a number of different formats. The generic I/O device format supports digital data by the individual bit, 8 bits, 16 bits, and 32 bits. Table 4 shows only the mandatory registers for 8-bit access.

A handy operation for digital I/O is to allow the polarity of the data to represent either the true or the inverted state of the inputs or outputs. The polarity index location determines this action for each corresponding bit. Other optional locations allow for changing input data to trigger a CAN change of state (COS) message. Outputs have an optional error mode that can define how they react to an error. For example, when enabled, the error mode can use the error value to place all of the outputs in a predetermined safe state.

Analog data can be provided in many formats as well. In addition to the 8-bit format shown in Table 4, the 16- and 32-bit formats are also supported. Provisions are made for floating-point or a manufacturer's specific format. Some scaling and offset factors are also available. Like the optional digital input COS function, an analog channel can produce messages based on its value compared to a number of parameters (<, >, =, and <>). Analog outputs also have the optional error mode associated with each output.

Index	Subindex	I/O	Index description	Subindex name	Data type
0x6000	0x00	Digital input	Read digital input	Number of 8-bit outputs	0x05
	0x01			Read inputs	0x05
0x6002	0x00		Digital input polarity	Number of 8-bit outputs	0x05
	0x01			Polarity of inputs	0x05
0x6200	0x00	Digital output	Write digital output	Number of 8-bit outputs	0x05
	0x01			Write outputs	0x05
0x6202	0x00		Digital output polarity	Number of 8-bit outputs	0x05
	0x01			Polarity of outputs	0x05
0x6206	0x00		Error mode output (optional)	Number of 8-bit outputs	0x05
	0x01			Error mode output	0x05
0x6207	0x00		Error value output (optional)	Number of 8-bit outputs	0x05
	0x01			Error Value Output	0x05
0x6400	0x00	Analog input	Read analog input	Number of 8-bit inputs	0x05
	0x01			Read input	0x05
0x6410	0x00	Analog output	Write analog output	Number of 8-bit outputs	0x05
	0x01			Write Outputs	0x05
0x6443	0x00		Error mode output (optional)	Number of 8-bit outputs	0x05
	0x01			Error mode output	0x05
0x6444	0x00		Error mode value (optional)	Number of 8-bit outputs	0x05
	0x01			Error value output	0x05

.able 4—I/O data is stored at specific locations in the device profile section of the object dictionary according to the I/O function. All of the available functions are explained in a document standard (DS) referenced by the device type at location 0x1000 (see Table 2). A flexible system of operators can help format and manipulate data.

SDO

I already covered the COB IDs used in CANopen and how the predefined IDs are used to handle various communications activities needed with any higher-level protocol based on a communication standard (the CAN interface in this case). I interrupted my description of PDO and SDO to describe the object dictionary (a table of various information). Knowing what's presented in this table should give you a clearer picture of just why you might need the SDO and PDO functions.

Let's start with the SDO (see Table 5, p. 76). Note that there is a transmit and a receive SDO. These implement generic access to every dictionary object. Using the SDO (COB ID + node address), every node (plus any network management interface) has access to everything. A CAN message with a COB ID of the Transmit Service Data Object (TSDO) is requesting data from a particular node. A CAN message with a COD ID of the Receive Service Data Object (RSDO) is providing data to a particular node. SDO transactions use a request/response format. Thus, every transfer requires two messages—a message to the TSDO or RSDO followed by the node's response—and could potentially hog bandwidth and time and should be used strictly for service data purposes. Enter the PDO.

FIRST PDO

The most obvious difference between the SDOs and the PDOs listed in the predefined COB ID set in Figure 1 is that there are multiple PDOs. This should give you a hint at their importance. Certainly, CANopen allows the use of SDO request/response transfers to get I/O data.

Let's suppose an I/O board has 16 digital input bits, 16 digital output bits, two 8-bit analog inputs, and two 8-bit analog outputs. To gather this data (based on the 8-bit data format of Table 4), you need four separate SDO transfers (eight messages) to complete the transaction. Using a PDO, all of the data can be mapped into one PDO location (a single message). The mapping of data shown in Table 3 packs up to 64 bits of data in a single PDO. This data can be packed in various formats as long as it doesn't exceed 64 bits (8 bytes).

A node that monitors temperature, for instance, doesn't need to know which nodes (if any) require this information. It may set up its first TPDO to send out a message announcing the temperature periodically or on a 2° change in temperature. Every node that is interested in this data needs to somehow pay attention to that Transmit Process Data Object (TPDO) to receive the information without having to ask for it. This makes for efficient use of the available bandwidth. Any node can receive this data simply by redefining the COD ID of one of its Receive Process Data Objects (RPDO). Let's take a look at how this is accomplished.

You don't necessarily think of RAM when you think of a look-up table. But the basis of the CANopen dictionary is to use RAM so that entries (at least certain ones) can be modified on the fly. Let's say that node 1 is reporting temperature in a message via its TPDO. It uses a COB ID of 0x181 to transmit a message of up to 8 data bytes of packed data.

You may have noticed that the TPDOs and RPDOs have individual default locations, and a different one for each node. For other nodes to automatically receive and make use of the temperature information broadcasted by node 1, these nodes must have any of their RPDO COB IDs changed to 0x181. This links TPDO 0x181 to those nodes implementing an RPDO 0x181. Data originating at the temperature input on node 1 is automatically transferred into any node implementing an RPDO equal to the TPDO. In fact, the mapping of that RPDO also divvies up the packed data into specific I/O locations on that node. CANopen has tremendous flexibility.

NETWORK MANAGEMENT

If you manufacture two CAN nodes that are used exclusively on an internal two-node CAN network, you intimately know each node. And they can be stat-

Client SDO data byte 0								Byte 1–3	Byte 4–7
b7	b6	b5	b4	b3	b2	b1	b0		
Client command specifier			Download segment request						
0	0	0	0 Then toggle each segment	0–7 Number of unused bytes			Last segment 1=Yes 0=No	Data	Data
			Initiate download request						
0	0	1	x	If Expedite=1 and Data Size= 1, then 0–3 Number of unused bytes		Expedite 1=Yes 0=No	Data size indicated 1=Yes 0=No	Index, subindex	Data
			Initiate upload request						
0	1	0	x	x	x	x	x	x	x
			Upload segment request						
0	1	1	0 Then toggle each segment	x	x	x	x	x	x
			Abort transfer						
1	0	0	0	0	0	0	0	Index, subindex	Error code
Server SDO data byte 0								Byte 1–3	Byte 4–7
Server command specifier			Upload segment response						
0	0	0	0 Then toggle each segment	0–7 Number of unused bytes			Last segment 1=Yes 0=No	Data	Data
			Download segment response						
0	0	1	0 Then toggle each segment	x	x	x	x	x	x
			Initiate upload response						
0	1	0	0	If Expedite=1 and Data Size=1, then 0–3 Number unused bytes		Expedite 1=Yes 0=No	Data size indicated 1=Yes 0=No	Index, subindex	Data
			Initiate download response						
0	1	1	x	x	x	x	x	Index, subindex	x
			Abort transfer						
1	0	0	0	0	0	0	0	Index, subindex	Error code

Table 5—*SDO commands can transfer 4 data bytes using an expedited transfer (including the index and subindex) of the dictionary object being requested. When expedited, if segment transfers are supported, they may be used to speed up larger requests using a 7-data-byte format.*

ically configured (in the code) to work together without the need for any additional network management. You have performed all the network management necessary for these nodes to live happily ever after. However, all bets are off when you want to add a third-party node. With some universal CAN widget, you must probe its dictionary objects to discover all of its secrets and change any object parameters (like the RPDO just discussed) to make it compatible with the other nodes.

Wouldn't it be easier if each node came with a datasheet? Well, the CANopen standard requires this. Dictionary objects along with their data comprise an electronic datasheet (EDS). Although a physical datasheet can be created via SDO probing, every manufacturer is required to provide a copy of its corresponding datasheet. This datasheet is an integral part of the conformance testing in terms of EDS conformance and device reflection.

Even in the simplest of systems, it's likely there will be at least one node with some kind of upper-level management associated with it. It might be an embedded node that simply monitors heartbeats and indicates a problem using an LED output. Or it might be a programmable logic controller (PLC) capable of extensive status reporting. When the configuration of a system exceeds what can be handled manually, network integration tools become necessary.

Because much of the dictionary is RAM-based, CANopen configuration tools give you a means to configure each node dynamically in a way that integrates their functions with all of the other nodes in the system. Simulation tools can also help find deficiencies in the system's design before it's actually implemented in hardware.

OUT OF SIGHT

As designers, you must often make decisions about which direction to take when implementing a plan involving some communications between devices. Sometimes the advantages of employing a standard underlying technology like CAN makes sense. After you make that leap of faith, there are other challenges to confront. You'll need to create a data exchange format and determine how other problems will be handled. Why reinvent the wheel?

You may have good reason to not use some higher-level protocols. This usually has to do with the minimum requirements of those solutions. As a standard is tweaked to handle more possibilities, it can grow to unbelievable proportions. Just handling the minimum requirements may not even fit into some microcontrollers. The CANopen protocol is powerful, yet the minimum requirements allow it to be implemented in all but the smallest microcontrollers. Implementing it can open the door to a world of opportunity for your widget. Even if you are developing a system consisting of in-house devices, consider providing a path with a future. Get connected today and be ready for tomorrow. ■

Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for Circuit Cellar *since 1988. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circuitcellar.com.*

RESOURCES

CANKingdom information, www.cankingdom.org.

CiA, "CANopen, An Overview," 2005, www.canopen.org/canopen/index.html.

———, "CANopen Device Profile Generic I/O Modules," DS401 V. 2.1, www.canopen.org/downloads/cia specifications/?1137.

———, CANopen product guides, www.canopen.org/products/pg2005/html/index-116.htm.

Open DeviceNet Vendor Association, www.odva.org.

O. Pfeiffer et al., *Embedded Networking with CAN and CANopen*, RTC Books, 2003.

O. Pfeiffer, "Embedded Networking with MicroMessaging," *Circuit Cellar* 159, October 2003.

———, "Implementing CANopen," *Circuit Cellar* 161, December 2003.

SAE International, SAE J1939 Standards, www.sae.org/standardsdev/groundvehicle/j1939.htm.

anual says the
25 V) at up
nd seems
spec.
ply,

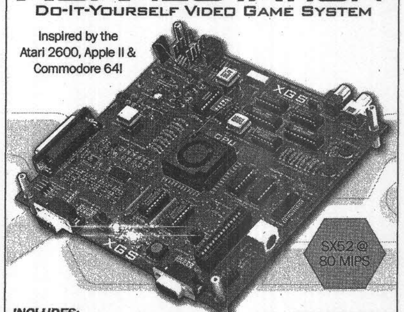
NEW! CAN FX

USB to CAN Interface
USB 2.0 Hi-speed 480Mbps!
2006 marks our 10th year of selling CAN interfaces in over 30 countries!
We're celebrating by offering our newest and fastest interface at only
$149 USD.

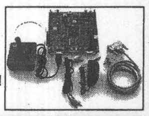
Zanthic Technologies Inc.
403-526-8318
www.zanthic.com

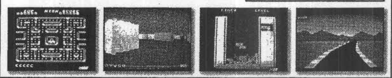
ANDRE LAMOTHE'S
XGAMESTATION
DO-IT-YOURSELF VIDEO GAME SYSTEM

Inspired by the Atari 2600, Apple II & Commodore 64!



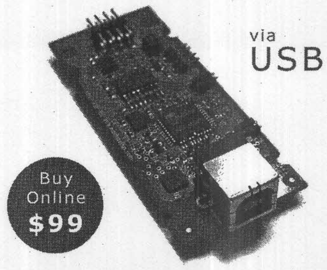
INCLUDES:
- Assembled XGS Micro Edition Unit!
- Complete Development Kit!
- Tools, Demos & Utilities!
- eBook on Designing the XGS Console!
- Cables and Power Supply Included!





WWW.XGAMESTATION.COM
(925) 736-2098 SUPPORT@NURVE.NET

Buy a CAN interface for your PC!
via USB



Buy Online $99

- Windows, Mac OS X & Linux support
- CAN monitor & software libs included
- OEM ready (volume discounts)
- USB 2.0 full speed
- Powered from USB bus or external
- CAN 11 & 29 bit ids, 1Mbit/sec

Machine Bus Corporation
www.machineBus.com
866 864-8100 (toll free)